嵌入式应用中的循环级线程推测并行性分析 *

卜得庆 [1a, 1b]，王耀彬 [1a, 1b†]，李　凌 [1a, 1b]，杨　洋 [2]，程一鸣 [1a, 1b]，刘志勤 [1a, 1b]，吴亚东 [1a, 1b]

(1. 西南科技大学 a. 计算机科学与技术学院; b. 四川省军民融合研究院，四川 绵阳 621010; 2. 四川省计算机研究院，成都 610041)

摘　要：如何有效利用多核提供的丰富晶体管资源对串行程序的执行进行加速是当前研究中的热点问题。线程级推测技术（thread-level speculation，TLS）旨在充分利用多核资源，最大化地开发出串行代码中存在的潜在并行性。目前 TLS 技术已经在多种串行应用的并行化工作中得到有效利用，但嵌入式应用程序仍未在推测并行化方面进行有效的分析。因此，本文选取了 8 个具有代表性的嵌入式应用，对其在循环级推测并行化中的性能提升潜力和运行时特征（数据依赖、线程粒度和并行覆盖率）进行探讨。实验结果表明：a）利用线程级推测并行化嵌入式应用的加速效果优于指令级并行技术，实验中的最大加速比达到了 13.29；b）在嵌入式应用领域，该技术可以有效利用 4 到 8 核的计算资源。

关键词：线程级推测；多核；嵌入式应用；数据依赖

中图分类号：TP302.7　　doi: 10.3969/j.issn.1001-3695.2018.04.0244

Profiling loop-level speculative parallelsim in embedded applications

Bu Deqing[1a, 1b], Wang Yaobin[1a, 1b†], Li Ling[1a, 1b], Yang Yang[2], Cheng Yiming[1a, 1b], Liu Zhiqin[1a, 1b], Wu Yadong[1a, 1b]

(1. *a. School of Computer Science & Technology*, *b. Sichuan Civil-military Integration Institute*, *Southwest University of Science & Technology*, *Mianyang Sichuan* 621010, *China*; 2. *Sichuan Institute of Computer Sciences*, *Chengdu* 610041, *China*)

Abstract: How to effectively utilize the rich transistor resources provided by multi-core to accelerate the execution of serial programs is currently a hot issue in research. Thread-Level Speculation (TLS) aims at making full use of multi-core resources and maximizing the potentially parallelism in serial code. At present, TLS technology has been effectively utilized to parallelize several serial applications. However, speculative thread level parallelism in embedded applications has not yet been explored thoroughly. Therefore, eight selected representative embedded applications were analyzed in this paper from their potential parallelism and runtime characteristics(data dependency, thread granularity and parallel coverage) in loop-level speculation. The experimental results show that: (1) for embedded applications, the speculative thread level parallelism is better than that in instruction level parallel technology, and the maximum speed up in experiment achieves 13.29. (2) In the field of embedded application, the technology can effectively utilize resources of 4 to 8 core computing.

Key words: thread-level speculation; multi-core; embedded application; data dependency

0　引言

随着多核时代的来临，如何有效利用处理器核上的丰富晶体管资源是当前研究的热点问题。由于超标量技术已很难再挖掘并行性，使得线程级并行（thread-level parallelism，TLP）技术得到了迅猛发展，且该技术特别适用于片上多核芯片（chip multiprocessor，CMP）[1~3]。目前，大部分应用程序还是根据单核处理器所设计的，若要在 CMP 上执行则需要重新编写该程序的大部分内容。线程级推测（TLS），或称推测并行化（speculative parallelization，SP）作为开发串行程序中线程级并行性的关键方法，允许多个线程在不同核上推测执行，以提高程序的并行效果[4]。

由于早期评测处理器性能的指标是比较其每秒运行指令条数的多少（Million Instructions Per Second，MIPS），其衡量标准在不同指令集结构上很难统一。而 20 世纪 80 年代提出的 benchmark 基准测试程序集评测方法，以其衡量机器性能标准

收稿日期：2018-04-02；修回日期：2018-05-29　　基金项目：国家自然科学基金面上项目（61672438）；国家重点研发计划资助项目（2016QY04W0801）；四川省军民融合研究院开放基金资助项目（2017SCII0213）；四川省教育厅研究项目（18ZB0603）；西南科技大学科研项目（17xn0045，17xn0038，17lzx621，13zx7101，17lzxt10）

作者简介：卜得庆（1993-），男，河南安阳人，硕士研究生，主要研究方向为计算机系统结构（568770592@qq.com）；王耀彬（1982-），男（通信作者），教授，博士，主要研究方向为计算机系统结构；李凌（1982-），男，讲师，博士，主要研究方向为网络安全、高性能计算、数值模拟与仿真；杨洋（1982-），男，工程师，硕士，主要研究方向为虚拟现实技术、软件工程；程一鸣（1991-），女，硕士研究生，主要研究方向为计算机应用技术；刘志勤（1962-），女，教授，硕士，主要研究方向为高性能计算；吴亚东（1979-），男，教授，博士，主要研究方向为科学计算与可视化.

更为全面的优点受到了业界的广泛认可。MiBench 是 Michigan 大学推出的一套免费的嵌入式应用程序基准测试集合，该集合用于测试不同范围内的基准，其源代码公开，较多的用于学术研究[5]。因此，该测试集在推出后，便受到学术界的广泛认可。目前，对嵌入式应用程序 MiBench 的评测分析主要集中在优化 GCC 编译器[6]、机器学习算法[7]以及代码变异[8]等方面，并没有在线程级推测方面对其分析评测。

本文的目的是对嵌入式应用在循环级推测并行化中的性能提升潜力和运行时特征进行探讨。通过选取嵌入式应用程序 MiBench 中 8 个具有代表性的程序和 SPEC CPU 中的 bzip2，设计了一种程序动态剖析机制，包括记录内存地址的哈希结构等核心数据结构，用于分析程序在推测并行化中的依赖特征、线程粒度、可并行化覆盖率和核心数量对加速比的影响，并根据影响因素在硬件资源和程序性能方面找到一个平衡点。

1 相关工作

1.1 典型 TLS 方案

Multiscalar[9]技术方案，虽然没有明确提出 TLS 的概念，但其执行过程与目前所用 TLS 的技术方法极其相似;Hydra[10]技术方案以软硬件结合的方式，实现了 TLS 机制并简化了硬件系统的设计，推动了 TLS 的发展。而目前，Wang 等人[11]提出一种在写操作上复制的 cache 的 TLS 新模型，使通用应用程序加速比平均提高 5.69 到 10.04 倍；Cao 等人[12]提出一种自动选择最好缓冲机制来提高 TLS 的性能，而且大部分程序的性能可以提高 75%以上；本文的前期工作[13]也对反向神经传播网络等应用的线程级推测并行性进行了深入分析。综上所述，线程级推测技术已被有效利用在多种类型的串行应用并行化工作中。

1.2 嵌入式应用程序的相关优化

目前，关于 MiBench 的加速评测主要有：文献[6]提出的一种优化过 GCC 编译器的架构，提高嵌入式应用程序 MiBench 的加速比，并通过编译器优化选择算法和高级混合消除算法与不同层次的优化算法进行对比；文献[7]中提出用机器学习算法，通过降低嵌入式应用程序 MiBench 的时间和空间复杂度来提高程序加速比，并与文献[6]进行对比；文献[8]中提出一种新颖算法——代码变异，将应用程序转换成复杂的逆向工程，使应用程序的动态存储和控制流行为发生转变，进而转变其程序性能。但目前，MiBench 仍未在线程级推测方面进行有效分析。

2 推测模型和剖析机制

在芯片设计的初始阶段，硬件仿真作为目前最有效的技术手段，能够基于特定硬件系统模型进行全方位的性能评测，进而对芯片设计中的各项关键性能影响因素进行分析和权衡。和直接进行硬件系统开发相比，仿真不仅可以高精度地完成硬件设计方案地有效性验证工作，而且可以低开销地进行各种硬件设计方案的全面比较和调整。因此，本文设计了如下的推测模型和剖析机制对串行代码中的推测并行性进行分析。

2.1 推测模型

TLS 技术可以将程序中的单个线程划分为多个可自动并行执行的多线程。TLS 的执行过程为：顺序执行第一个线程，推测执行其余线程，根据原来串行程序的提交顺序进行提交，若在程序的并行执行过程中发生冲突，则撤销冲突线程的执行，执行回滚操作重新执行该线程[14]。如图 1 所示，在图 1（a）中，线程在核上顺序执行，这是串行程序目前仍然使用的执行过程。在图 1（b）中，程序在运行时，系统将其分解为许多线程，并将这些线程分配到各个核中。当开始执行的时候，第一个核通知其余核加载程序和变量，并分发开始的信号给其余核。当其它核收到第一个核的信号后，便开始推测执行。若第一个核提交完成指令后，则由第二个核继续提交计算值，以达到顺序执行效果。若最后一个线程执行完并提交最后计算值的时候，便由该执行核发出程序结束的信号给其余核。

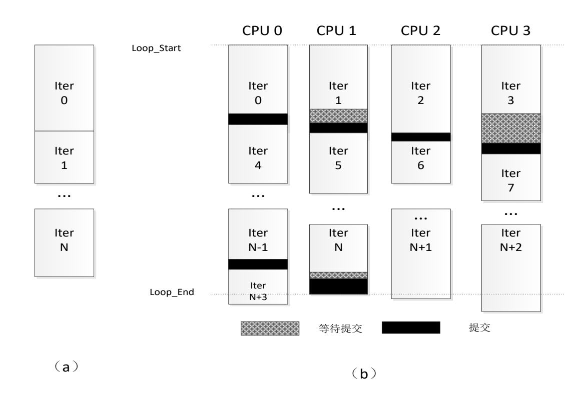


图 1 传统程序和推测线程级程序运行模型

TLS 的线程划分方案常常以程序的控制流特征为依据，选取程序中的循环结构和子程序结构[15]。从系统结构方面看，程序的本质是由许多机器指令构成，其主要结构包括: 顺序结构和转移结构，循环控制是转移结构中的核心部分，其性能也是程序执行性能的关键[16]。目前，并行化的主要对象为程序中的循环结构[17]，因此本文将程序的循环作为研究重点。在本工作中，程序循环的选择即可并行化区域覆盖率是通过 GNU Prof 工具收集过去程序运行时的信息，根据收集到的信息选择时间占比比较大的子程序中的循环。

线程粒度和数据依赖是 TLS 的关键影响因素[18]。线程粒度指一个线程所包含的动态指令数。若线程粒度过小，则分发的线程数量增多，因此线程的各种操作将带来过多开销，抵消推测并行的效果；当线程粒度过大，则分发的线程数量减小，推测的缓存数据增多，因此依赖冲突的概率增大，进而降低并行度。数据依赖指子线程与父线程的引用值之间存在某种依赖关系。因此，文献[19]中提出生成者与消费者的距离，并根据此比值得到线程间数据依赖特征。其主要思想如图 2（a）所示，令 A 为生产者距离（程序在最后一步写到程序开始的距离），B 为消费者距离（程序第一次读到程序开始的距离），α=A/B（生产者距离/消费者距离）。当 α>1 的时候，说明并行情况良好，就会产生如图 2（b）一样；若 α<1，其与串行程序执行相似，如图 2（c）所示。在本文中，将 α <1 的依赖关系称之为致命依赖，

将 1<α<2 称为危险依赖，将 α>2 称为安全依赖。

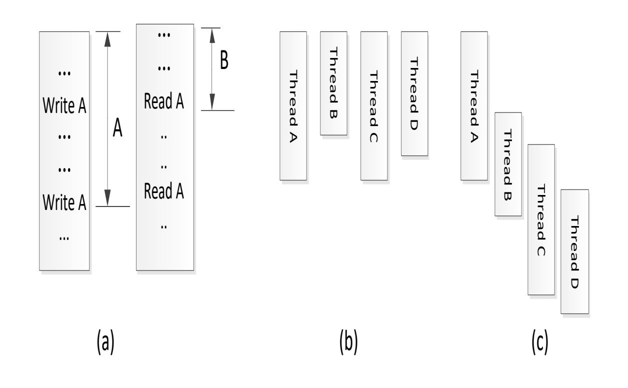


图 2 消费者距离与生产者距离,理想情况下与非理想情况下并行模型

2.2 剖析机制

在本工作中，针对循环的剖析工具其设计思路为：a）选取程序中具有一定的计算量且有较高的可并行化区域为“热点”代码片段，将其作为初步的候选者;b）再针对这些热点区域进行剖析，对影响性能的关键因素（数据依赖、线程粒度和并行覆盖率）进行量化分析;c）输出量化分析结果。

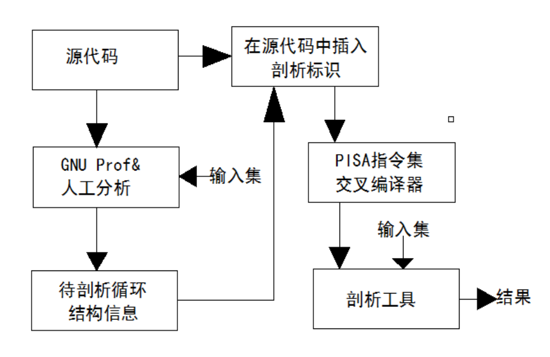


图 3 剖析工作具体流程

如图 3 所示，首先利用 Linux 系统自带的 GNU Prof 工具对程序进行初步分析。该工具可粗略地分析出程序运行时各个子程序运行的时间比例，以及函数调用关系等。通过此工具提供的信息，选取其中时间占比超过 3%的热点片段，将其作为初步的推测候选区域；然后在候选区域中选取循环并插入剖析标识，经编译器进行交叉编译得到可执行的二进制代码文件；最后，通过识别汇编文件中指令的手段，使剖析工具自动识别二进制代码文件中的热点区域，对其进行剖析，并根据影响性能的关键因素进行量化分析，得出剖析结果。如图 4 所示，在自动识别剖析标识中，利用支持 PISA 指令集的反汇编工具 objdump4pisa，将二进制代码反汇编成汇编代码，loop_id 被编译保存在 4 号寄存器中，循环变量的地址被编译并保存在 5 号寄存器中。此后，剖析工具通过识别汇编文件中的 jal 指令和循环标识，对热点区域进行剖析。

根据剖析流程，本文设计的循环剖析原理核心数据结构如图 5 所示。若程序中包含多个循环，则每个循环被分解为 n 个 iteration_list_entry_t，且每个 iteration_list_entry_t 被分解为迭代次数 N 个的 iteration_t 结构，Hash_list 则是用来记录对储存器写入操作的数据结构，根据 Hash_list 中的 iteration_list_head 和 iteration_list_tail 可以计算生产者和消费者距离。在 iteration_t 中有循环展开中的唯一循环 id，开始时间，以及下一次循环展开地址的指针和上一次循环展开地址的指针，hash_next 记录的是下一次循环展开对存储器操作的地址指针，hash_pre 记录的是上一次循环展开对存储器操作的地址指针。当剖析部分的后继代码读取到某一存储单元时，会被剖析工具拦截，剖析工具通过检索该结构的存储单元可以得到最后一次写时间，如果当前时间小于最后一次写时间，则将当前时间设置为该存储单元地址的最后一次写时间。同时，系统还会保存一个串行版本的运行时间，这两个时间的比值就是想要得到的加速比。

```
...
for(i=0;i<N;i++)
{
   loop_start(0,&i);
  /*开始标识*/
       ...
}
loop_end();/*结束标
             识*/
...
```

```
$L2:
   lw  $2,16($fp)
   lw  $3,N
     ...
   j   $L3
$L5:
  move  $4,$0
  addu  $5,$fp,16
  jal  _Iteration_start
     ...
   jal   _Itertion_end
$L4:
    lw    $3,16($fp)
   addu  $2,$3,1
   move  $3,$2
   sw    $3,16($fp)
   j     $L2
$L3:
      ...
```

图 4 剖析工具中循环标识示意图

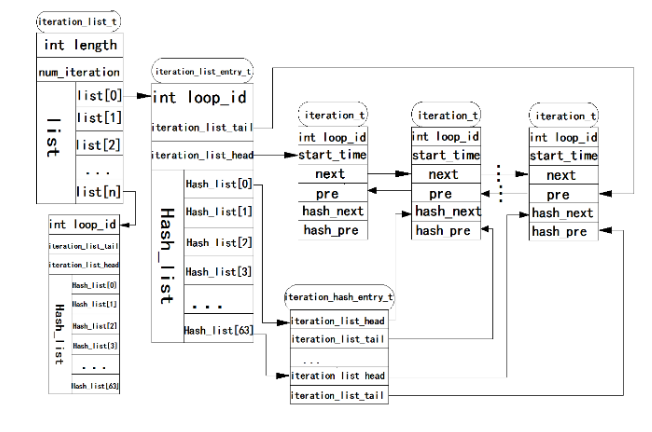


图 5 循环剖析核心数据结构示意图

3 实验数据及分析

在本工作中，实验所采用的环境具体描述如表 1 所示。

表 1 实验环境说明

实验平台	指令集	模拟器	编译器
基于 Linux 的 Ubuntu 12.04 开发版本	基于 Simplescalar 工具集的 PISA 指令集	基于 Simplescalar 工具集中的 sim-fast 修改扩充版本	采用 Simplescalar 工具集中提供的经过后端改造的 gcc-2.7.2.3

在实验中，之所以选择 simplescalar 工具集中的 sim-fast 模拟器进行修改扩充，是因为该模拟器每周期可以执行一条指令，其运行速度使剖析时间非常适应编译器的开销。

MiBench 中共含有 35 个基准程序，而本文选择了其中 8 个典型的应用进行分析，并且还选取了 SPEC 中的 Bzip2 进行剖析，便于与文献[6]进行数据对比。所选应用程序的详细列表如表 2 所示。

表 2 所选应用分类及应用说明

程序名	分类	应用说明
Dijkstra	Network	Dijkstra 算法实现
Bzip2	SPEC	无损压缩算法实现
Bitcount	Automotive	统计一个整数数组包含的 bit 中 1 的个数
Susan	Automotive	图像识别工具包
Jpeg	Consumer	JPEG 编解码程序
Patricia	Network	Patricia Trie，用于叶子稀疏的树结构
Blowfish	Security	blowfish 加密/解密算法

3.1 时间对比

本文最佳运行时间(mm)与文献[6]中最佳时间(mm)的对比，如表 3 所示，表中 A 为该程序在无加速情况下运行时间，B 为文献[6]中最佳加速效果运行时间，C 为本文最佳运行时间。

表 3 时间对比

程序	A	B	C
Bzip2	8.81	5.28	5.03
Bitcount	2.06	--	0.79
Susan	14.18	7.16	6.92
Jpeg	9.59	6.81	4.38
Dijkstra	0.99	0.69	0.07
Patricia	4.52	2.99	3.02
Blowfish	14.8	8.63	7.51

从表 3 可以看出，除了 network 的 Patricia 的运行时间略小于文献[6]中运行时间，其他程序运行时间均小于文献[6]中的最佳运行时间，由此表可以证明本工作中运用的线程级推测技术产生的结果要好于优化 GCC 编译器所产生的结果。

3.2 加速比分析

3.2.1 Dijkstra

对 Dijkstra 程序剖析所产生的加速比如图 6 所示。随着核数的不断增加，该程序的加速比呈线性增长，而到了 64 核之后，加速比将平衡在 13.10 到 13.29 之间。通过分析源代码发现程序的循环之间确实不存在数据依赖，并且该程序的可并行化区域很高，安全依赖占比很大，所以该程序的加速比会随着核数的增加不断提高，但到了 64 核便达到了该程序的推测并行化限度。

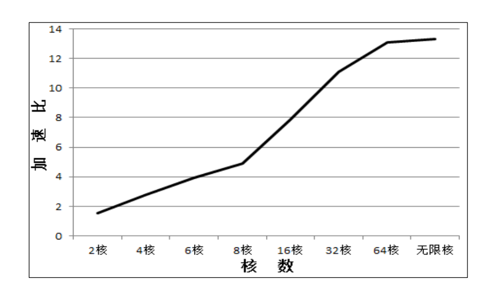


图 6 dijkstra 在不同核数条件下的加速比

3.2.2 Bzip2

使用模拟器对 Bzip2 程序进行剖析，得到的加速比如图 7 所示。通过对程序循环进行线程级并行分析，发现程序在 4 核和 8 核时，加速比增加了 0.39 和 0.48。随着硬件资源的继续增加，并与 4 核和 8 核的加速比对比发现，4 核和 8 核的加速比已经相对趋于稳定，虽然加速比也缓慢增加，但却浪费了硬件资源。

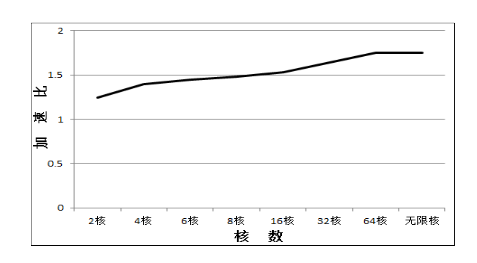


图 7 Bzip2 在不同核数条件下的加速比

3.2.3 Bitcount

对 Bitcount 程序的剖析结果所得到的加速比从图 8 中可以看出，当核数由 2 核逐渐增加到 8 核之前，所得到的加速比成线性增长，但 8 核之后，加速比不在提高。由此可见，合理利用核心数，是一个举足轻重的问题。

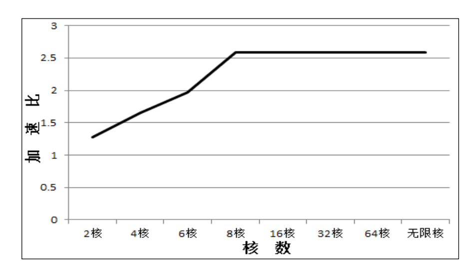


图 8 Bitcount 在不同核数条件下的加速比

3.2.4 Susan

从图 9 中可以看出，当核数由 2 核逐渐增加 32 核之前，所得到的加速比呈缓慢线性增长趋势，而在 4 核和 8 核时，加速比分别提高了 60%和 80%。结合源代码分析发现该程序主要是对图像的处理，基本处理都是逻辑判断。因此，循环不多导致可并行化区域不高，所以该程序的加速比不会太高。

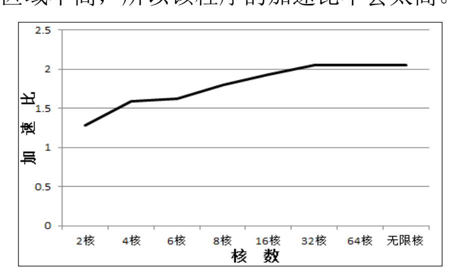


图 9 Susan 在不同核数条件下的加速比

3.2.5 Jpeg

从图 10 中可以看出，程序 encode 和 decode 的加速比在 16 核之前都不高，在 16 核之后也仅仅是 encode 程序的加速比有所提高。由于该程序是压缩与解压的应用，涉及许多指针类型的数据，所以数据依赖较为严重。即使不断增加核数，也很难得到理想的加速比。

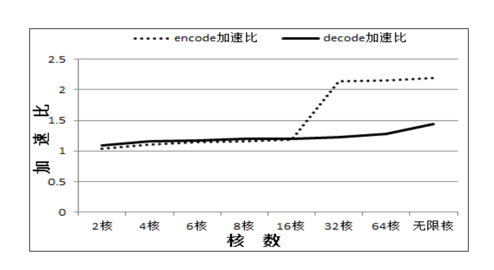


图 10　jpeg 在不同核数条件下的加速比

3.2.6 Patricia

对 Patricia 剖析所产生的加速比如图 11 所示。该程序不适合进行推测多线程处理，对其源码分析发现，该算法的实现多是采用指针类型进行编写，导致循环和后继代码之间数据依赖非常严重，所以加速比趋于稳定。

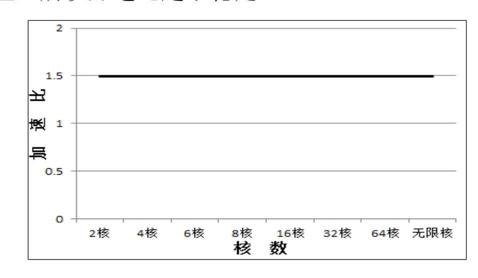


图 11　patricia 在不同核数条件下的加速比

3.2.7 Blowfish

对 blowfish 剖析所产生的加速比如图 12 所示。程序 encode 和 decode 的加速比在 8 核之前相对比较稳定，而在 8 核之后加速比才开始上升，并且到了 64 核其加速比才增加了 1 倍。因为核数的增加，其成本也在无限增加，所以在核数与加速比之间找到一个平衡点非常重要。当然由于程序 encode 和 decode 所存在的线程粒度、可并行覆盖率和数据依赖等问题基本一致，所以产生的结果也如出一辙。

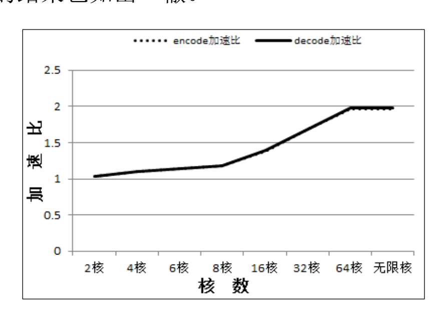


图 12　blowfish 的 encode 和 decode 在不同核数条件下的加速比

3.2.8 数据分析

由于各程序所涉及到的影响因素各不相同，得到的加速比也截然不同。为了找出更多的规律，将实验得到加速比做了归一化处理，即分别以各种情况下的加速比与 64 核所得到的加速比相比，如图 13 所示。

从图 13 可以看出，除程序 dijkstra 不适合利用线程级并行加速之外，其余程序均随着核数的增加，加速比增长率呈缩小趋势；极端的情况如 Bitcount，当核数大于 4 时，加速比增长率趋于稳定；大部分程序在 4 核到 8 核之间时，其程序加速比达到程序性能与硬件资源的平衡点。

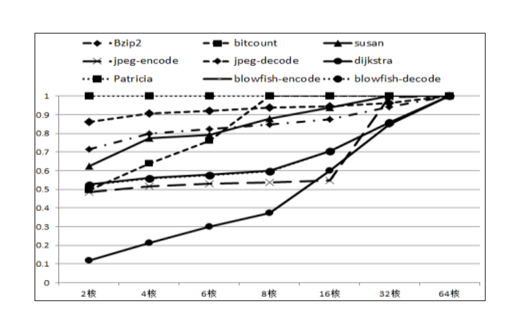


图 13　各程序归一化后的加速比变化曲线

3.3　影响因素分析

由于线程粒度、并行化覆盖率和依赖特征都是线程级推测的影响因素。因此，后续部分分析了所选应用程序的线程粒度、可并行化覆盖率和依赖特征对加速比的影响。

线程粒度情况如图 14 所示，大部分程序的线程粒度介于 10^6 到 10^8 之间，由于大部分嵌入式应用程序的推测线程粒度比通用应用中的大，所以针对嵌入式应用的推测多核处理器设计中，推测缓存的设计容量应比通用设计提高一个量级，以避免发生缓存溢出的现象。

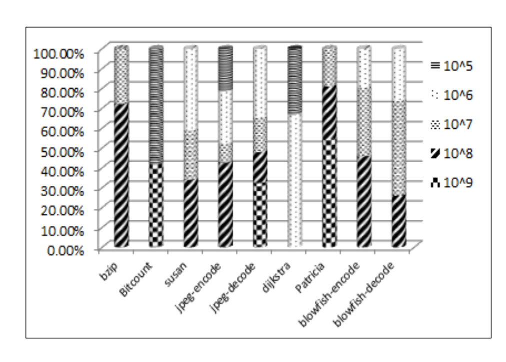


图 14　所选应用程序的线程粒度

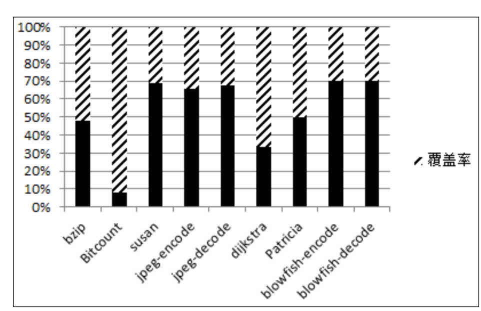


图 15　所选应用的可并行化覆盖率

可并行化覆盖率如图 15 所示，大部分程序的覆盖率介于 30%到 50%之间。而根据阿姆达尔定律，若可并行化的覆盖率越高，其程序才有利于线程级推测的开发。因此，并行区域不高也是导致加速比不高的一个因素。

依赖特征 α 的分布如图 16 所示，大部分程序的致命依赖在 50%以上，其中 3 个程序更是达到了 90%左右。当然，若 α >1 的程度越高即安全依赖越高，越有利于线程级推测的开发。因此，导致大部分嵌入式应用程序加速比不高其主要原因还是因为致命依赖较为严重。

结合源代码和性能因素影响分析，发现造成嵌入式应用程序并行加速比不高的原因主要有：首先，程序循环迭代的线程粒度较大而导致线程之间的数据发生的依赖冲突概率增大，这

也是造成所选应用程序加速比不高的主要原因。因此，在编写程序时，应注意拆分比较大的循环体，从而降低线程粒度；其次，大部分程序的可并行化区域覆盖率不高，且其致命依赖较高。为了提高程序循环的并行化限度，降低致命依赖，在编写代码时，应特别注意在循环体中尽量少使用指针型数据，或者使用其它数据替代掉指针数据，从而使程序在循环级线程并行取得更大效益，进而提高程序的并行化限度。

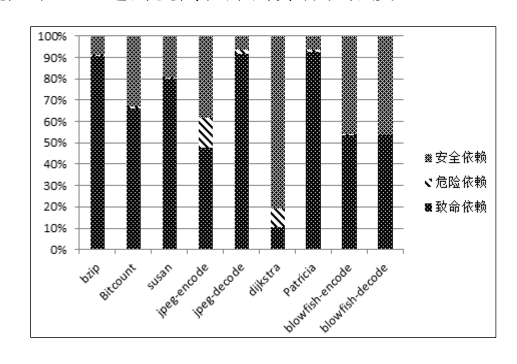


图 16　所选应用的依赖特征

4　结束语

通过对 MiBench 中的大部分程序以及 SPEC 中的 Bzip2 程序进行线程级推测并行化所得到的加速比进行比较和分析，得到如下结论：

a)利用线程级推测对嵌入式应用程序的加速效果要优于优化过的 gcc 编译器所产生的加速效果。因此，在多核时代，采取线程级推测技术可以对嵌入式应用程序起到很好的加速效果。

b)在 MiBench 中，当核数增加到 4 和 8 之间时，大部分程序的加速比可达到 1.65 到 4.89 之间，加速比性能提高 63.8%到 84.5%之间。因此，当面向嵌入式应用设计多核处理器时，核心的数目在 4 和 8 之间时，即可以提高硬件利用率，也可以使程序的加速效果达到最佳。

c)数据依赖是影响循环级线程推测的主要因素。在循环迭代中，如果适当的修改循环体或者循环控制中的变量，可以消除对应的迭代之间的数据依赖或者控制依赖，或者提高 α （消费者距离/生产者距离）比值，可以提高程序循环部分的可并行化限度，得到更高的加速比。

由于硬件仿真处于芯片研发的第一阶段，只有通过了该阶段的准入式评估，才能进入后续的一系列更精准的硬件原型验证和真实硬件系统评测等阶段。因而本文所提出的方案着眼于得出并行加速比理论极值，与实际硬件系统运行（特别是处理器访存系统工作情况）相比仍然存在一定的差异，后续还将对这些嵌入式应用进行寄存器级硬件原型仿真，进而得到更精准的评测数据。同时在未来的工作中，也将进一步剖析并分析该基准测试程序在推测子程序结构时的线程级推测可并行化限度，并与线程级循环推测的可并行化限度进行对比，从而为嵌入式推测多核芯片的研发提供设计依据。

参考文献：

[1] Luo Yangchun, Zhai A. Dynamically dispatching speculative threads to improve sequential execution [J]. ACM Trans on Architecture and Code Optimization, 2012, 9 (3): 1-31.

[2] Schmidt T, Liu Guantao. Exploiting thread and data level parallelism for ultimate parallel systemc simulation [C]// Proc of the 54th Annual Design Automation Conference. New York: ACM Press, 2017: articleNo. 79.

[3] Cai Yong, Li Guangyao, Liu Wenyang. Parallelized implementation of an explicit finite element method in many integrated core (MIC) architecture [J]. Advances in Engineering Software, 2018, 116: 50-59.

[4] Salamanca J, Amaral J N, Araujo G. Using hardware-transactional-memory support to implement thread-level speculation [J]. IEEE Trans on Parallel & Distributed Systems, 2018, PP (99): 1-14.

[5] Guthaus M R, Ringenberg J S, Ernst D, *et al.* MiBench: a free, commercially representative embedded benchmark suite [C]// Proc of IEEE WWC. Austin Texas: IEEE Press, 2001: 3-14.

[6] Andrews. J. Analysis of Mibench Benchmark Applications Using GCC Compiler [C]// Proc of the 3rd International Conference on Computer Science and Information Technology. 2013: 34-37.

[7] Andrews J, Sasikala T. Evaluation of various compiler optimization techniques related to mibench benchmark applications [J]. Journal of Computer Science, 2013, 9 (6): 749-756.

[8] Ertvelde L V, Eeckhout L. Dispersing proprietary applications as benchmarks through code mutation [C]// Proc of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 2008: 201-210.

[9] Vijaykumar T N, Sohi G S. Task Selection for a multiscalar processor [C]// Proc of the 31st Annual ACM//IEEE International Symposium on Microarchitecture. New York: ACM Press, 1998: 81-92.

[10] Hammond L, Hubbert B A, Siu M, *et al.* The Stanford Hydra CMP [J]. IEEE Micro, 2000, 20 (2): 71-84.

[11] Wang Qiong, Wang Jialong, Shen Li, *et al.* A software-hardware co-designed methodology for efficient thread level speculation [C]// Proc of the 17th IEEE International Conference on Computer and Information Technology. New York: ACM Press, 2017: 184-191.

[12] Cao Zhen, Verbrugge C. Reducing memory buffering overhead in software thread-level speculation [C]// Proc of the 25th International Conference on Compiler Construction. New York: ACM Press, 2016: 12-22.

[13] Wang Yaobin, An Hong, Liu Zhiqin, *et al.* Parallelizing Back Propagation Neural Network on Speculative Multicores [C]// Proc of the 22nd International Conference on Parallel and Distributed Systems. Piscatway, NJ: IEEE press, 2017: 902-907.

[14] 李翔. 面向多核的串行程序并行化关键技术研究 [D]. 西安: 西安理工大学, 2014. (Li Xiang. Research on the key technology for Parallel serial program based on multi-core [D]. Xi’an: Xi'an University of Technology,

2014.)

[15] Goossens B, Parello D, Porada K, *et al.* Computing on many cores [J]. Concurrency and Computation Practice and Experience, 2017, 29 (4): 1-20.

[16] 邹宇，薛小平，张芳，等. 用于程序循环控制的错误检测算法 [J]. 计算机应用, 2015, 35 (12): 3450-3455. (Zou Yu, Xue Xiaoping, Zhang Fang, *et al.* Error detection algorithm of program loop control [J]. Journal of Computer Applications, 2015, 35 (12): 3450-3455.)

[17] 李颖颖，庞建民，李雁冰，等. 一种面向众核处理器的嵌套循环多维并行识别方法 [J/OL]. 计算机应用研究, 2018, 35 (11) . (2017-11-10) [2018-03-10]. http://www. arocmag. com/article/02-2018-11-001. html. (Li Yingying, Pang Jianmin, Li Yanbing, *et al.* Multi-dimensional parallelism recognition method of nested loop for many-core processors [J/OL]. Application Research of Computers, 2018, 35 (11) . (2017-11-10) [2018-03-10]. http://www. arocmag. com/article/02-2018-11-001. html.)

[18] 刘斌，赵银亮，韩博，等. 基于性能预测的推测多线程循环选择方法 [J]. 电子与信息学报, 2014, 36 (11): 2768-2774. (Liu Bin, Zhao Yinliang, Han Bo, *et al.* A loop selection approach based on performance prediction for speculative multithreading [J]. Journal of Electronics and Information Technology, 2014, 36 (11): 2768-2774.)

[19] 梁博. 多核结构上的线程级推测关键技术研究 [D]. 合肥：中国科学技术大学, 2008. (Liang Bo. Study on the key technologies of thread-level speculation on multi-core platform [D]. Hefei: University of Science and Technology of China, 2008.)